Distributing Relational Model Transformation on MapReduce

Amine Benelallam, Abel Gómez, Massimo Tisi, Jordi Cabot

Distributing Relational Model Transformation on MapReduce

Amine Benelallam[a,*], Abel Gómez[c], Massimo Tisi[b], Jordi Cabot[d,c]

[a] *DiverSE team (Univ Rennes, Inria, CNRS, IRISA) 263 Avenue Général Leclerc, 35000 Rennes, France*
[b] *IMT Atlantique, LS2N (UMR CNRS 6004). 4, Rue Alfred Kastler, 44300, Nantes, France*
[c] *Universitat Oberta de Catalunya. Av. Carl Friedrich Gauss, 508860 Castelldefells, Spain*
[d] *ICREA. Passeig de Lluís Companys, 23, 08010 Barcelona, Spain*

Abstract

MDE has been successfully adopted in the production of software for several domains. As the models that need to be handled in MDE grow in scale, it becomes necessary to design scalable algorithms for model transformation (MT) as well as suitable frameworks for storing and retrieving models efficiently. One way to cope with scalability is to exploit the wide availability of distributed clusters in the Cloud for the parallel execution of MT. However, because of the dense interconnectivity of models and the complexity of transformation logic, the efficient use of these solutions in distributed model processing and persistence is not trivial.

This paper exploits the high level of abstraction of an existing relational MT language, ATL, and the semantics of a distributed programming model, MapReduce, to build an ATL engine with implicitly distributed execution. The syntax of the language is not modified and no primitive for distribution is added. Efficient distribution of model elements is achieved thanks to a distributed persistence layer, specifically designed for relational MT. We demonstrate the effectiveness of our approach by making an implementation of our solution publicly available and using it to experimentally measure the speed-up of the transformation system while scaling to larger models and clusters.

Keywords: Model Transformation; Distributed Computing; MapReduce; ATL; NeoEMF

*Corresponding author. Note this work has been carried out during the PhD thesis of Amine Benelallam within the AtlanModels team.

Email addresses: amine.benelallam@irisa.fr (Amine Benelallam), agomezlla@uoc.edu (Abel Gómez), massimo.tisi@inria.fr (Massimo Tisi), jordi.cabot@icrea.cat (Jordi Cabot)

1. Introduction

Model-Driven Engineering (MDE) has been successfully embraced in several domains for automating software development and manufacturing maintainable solutions while decreasing cost and effort. Indeed, recent work has shown the benefits of MDE in applications for the construction industry [1] (for communication of building information and inter-operation with different tools and actors), modernization of legacy systems [2], learning and Big Data analytics [3]. The AUTomotive Open System ARchitecture (AUTOSAR) used in the development of automotive software systems, and the Building Information Modeling (BIM [1]) are two successful standards involving MDE development principles in the software lifecycle for more than ten years and twenty years respectively. Model query and transformations are key operations to guarantee these benefits.

A model transformation (MT) is an operation responsible for translating one model to another. A complex transformation written in one of the general purpose languages (GPLs) can be extremely large and unmaintainable. Fortunately, model query and transformation languages come to the rescue, having been designed to help users in specifying and executing model-graph manipulation operations efficiently. They provide adequate constructs and facilities to specify modular and reusable transformations with less effort. The relational paradigm is the most popular among MT languages, based on the declarative definition of rules relating input and output model elements. Relational model transformations use tracing mechanisms to instate these relations.

Despite all these promises, the MDE approach is not widely applied to large-scale industrial systems. This is mainly due to the serious scalability issues that the current generation of MDE tools is facing [4, 5, 6]. Indeed, according to existing empirical assessments from industrial companies adopting MDE [4, 7], the lack of tools and technologies supporting collaboration and scalability are the substantial disadvantages in MDE. Such large systems exhibit a growing complexity in design, as well as the need to handle an increasing amount of data. For example, BIM contains a rich set of concepts (more than eight hundred) for modeling different aspects of physical facilities and infrastructures. A building model in BIM is typically made of several gigabytes of densely interconnected graph nodes. The model has been growing in time by incrementally integrating different aspects and stakeholders [8]. The AUTOSAR size has been similarly growing [9, 10]. Therefore, there is a calling need to develop a new generation of tools capable of coping with large models.

1.1. Problem statement

Foremost, MT operations involve mainly graph matching and traversing techniques. Such operations are CPU-consuming, which raises serious scalability issues as graphs grow in scale and complexity. Consequently, graph processing problems in general, and MTs in particular, can exceed the resources of a single machine (CPU). For example, in our experiments, we show how typical MT tasks in the reverse-engineering of large Java code bases may take several hours to compute in local.

One way to overcome these issues is exploiting distributed systems for parallelizing model manipulation (processing) operations over computer clusters. This is made convenient by the recent wide availability of distributed clusters in the Cloud. MDE developers may already build distributed model transformations by using a general-purpose language and one of the well-known distributed programming models such as MapReduce [11] or Pregel [12]. However such development is not trivial, especially since distributed programming requires familiarity with concurrency and distribution theory that is not common among MDE application developers, in particular when the set of possible execution paths can be large, and transformation result can be non-deterministic. Distributed programming also introduces a completely new class of errors w.r.t. sequential programming, which is linked to task synchronization and shared data access. Finally, it entails complex analysis for performance optimization, for instance, balancing computation load, and maximizing data locality. To summarize, we argue that the growth in data and complexity that is being experienced by the industry is ahead of the current generation of MT tools. This hampers the adoption of MDE in industrial contexts. Therefore, a new generation of MT engines should be provided.

1.2. A distributed platform for relational MTs

In previous work, Clasen et al. [13] draw the first lines towards a conceptual framework for handling the transformation of very large models (VLMs) in the Cloud. Their vision includes essentially two bricks, a *model transformation engine* and a *model access and persistence framework*, discussing different possible alternatives. In this paper, we provide a practical solution for these bricks, opting for the data-distribution scheme.

We introduce a distributed platform for running relational MT in the Cloud. We show that relational MTs, thanks to their specific level of abstraction, can be provided with semantics for *implicit distributed execution*. While we use the rule-based language ATL (AtlanMod Transformation Language [14]) to exemplify our approach, our distribution approach is applicable to the class of relational model transformation languages. We show that thanks to the properties of the ATL language, interactions among rule applications are reduced. The distribution in our proposed framework is implicit, i.e. the syntax of the MT language is not modified and no primitive for distribution is added. Hence developers are not required to have any acquaintance with distributed programming. The semantics we propose is aligned with the MapReduce computation model, thus, showing that rule-based MT fits in the class of problems that can be efficiently handled by the MapReduce abstraction.

Distributed computation models like MapReduce are often associated with persistence layers for accessing distributed data. In our second contribution, we propose a new model-persistence backend, NeoEMF/Column, that provides support for transparently decentralized persistence and access on top of a distributed column store. NeoEMF/Column provides lightweight serialization/deserialization mechanisms of data communicated across the network and concurrent read/write from/to the underlying backend.

We demonstrate the effectiveness of the approach by making an implementation of our solution publicly available[1] and by using it to experimentally measure the speed-up of the transformation system while scaling to larger models and clusters, and more complex transformations. To do so, we use two well-known model transformations, ControlFlow2Dataflow and Class2Relational. The paper illustrates in detail the integration of two components introduced in our previous work [15, 16]. In particular, we extend this work by providing the big picture of the ATL-MR approach, contributing precise definitions and properties(Section 4), carefully analyzing ACID properties required by distributed MTs and implementing them on top of NeoEMF/Column (Section 6), and deepening on failure management and data distribution(Section 7). Finally, we evaluate the scalability and performance of ATL-MR on top of NeoEMF/Column (Section 8).

1.3. Outline of the paper

The rest of the paper is structured as follows. Section 2 discusses the main related work in scalable MT and persistence. This section highlights the main limitations of existing approaches and tools in terms of scalability, then positions our framework w.r.t. them. Section 3 describes the running case of the paper and uses it to introduce the syntax of ATL and its execution semantics. Later, it briefly outlines the main concepts of the MapReduce programming model. Afterwards, Section 4 gives an overview of our framework for distributed MTs. It starts by presenting some definitions and properties to simplify the understanding of our approach, then describes the different steps of the distributed transformation process. Section 5 extends our previous work on a distributed engine for MTs with ATL on MapReduce [15]. Section 6 introduces our distributed persistence framework and its support for the set of ACID properties guaranteeing consistent model transformations. Section 7 illustrates the integration of processing and persistence layer. Section 8 discusses the evaluation results of our solution when applied to two use cases with different complexity. Finally, Section 9 concludes our paper and outlines some future work.

2. Related Work

Model transformation operations may involve intensive read/write from/to models. These models are traditionally stored in a persistence backend. Hence, the performance of a MT engine could be highly impacted by the performance of the underlying persistence backend. For example, the latency due to read and write operations may severely affect the execution time of a MT. In this section, alongside approaches on scalable MTs, we present related work on scalable persistence. First, we introduce related work attempting to improve graph and model processing performance, namely, parallel and distributed model/graph transformation approaches. Then, we review related work on

[1] `https://github.com/atlanmod/ATL_MR/`

scalable persistence organized by types of persistence backends, File-based, SQL-based, and NoSQL-based. Finally, we discuss the limitations and drawbacks of existing MT approaches and examine the appropriateness of current solutions for model persistence to distributed MTs.

2.1. Distributed and parallel graph processing

Parallel and distributed graph transformation is a well-studied problem, and well-defined fundamentals have been proposed in several works. In parallel and distributed graph transformations, rule applications are structured in both temporal and spatial dimensions. Graphs are split into sub-graphs for local transformations, then joined again to form a global graph [17]. Two main families of approaches have been proposed, shared memory (parallel) and message passing (distributed). In what follows, we examine existing solutions in parallel and distribute MTs. Afterwards, we briefly discuss some alternative execution semantics for scalable transformations, then we present some high-level languages for distributed data-parallel computing.

2.1.1. Distribution for graph processing languages

Among distributed graph transformation proposals, a recent one is Mezei et al. [18]. It is composed of a transformation-level parallelization and a rule-level parallelization with four different matching algorithms to address different distribution types. In another work [19], Izso et al. present a tool called IncQuery-D for incremental query in the Cloud. This approach is based on a distributed model management middleware and a stateful pattern matcher framework using the RETE algorithm. The approach has shown its efficiency, but it addresses only distributed model queries while we focus on declarative transformation rules.

Two approaches map a high-level graph transformation language to the Pregel programming model [20, 21]. Krause et al. [20] proposed a mapping of a graph transformation language and tool, Henshin, to the "BSP model transformation framework" on top of the BSP model. The approach is based on a code generator that translates the graph transformation rules and transformation units into a Pregel program. The matching phase is split into a series of local steps. A local step inspects if a local constraint is satisfied and generates, extends, or merges a set of partial matches. A search plan generation algorithm is responsible for the generation of local steps.

In a similar approach, Tung et al. [21] implemented their own DSL for the specification of graph transformations. This DSL is also compiled into a Pregel program and executed over a distributed cluster. The DSL inherits most of its constructs from UnCAL, a query language and algebra for semi-structured data based on structural recursion. The semantics of UnCAL was improved to support the basic Pregel skeletons. In contrast to the previous approach, this one supports successive applications of queries/transformations. In particular, both approaches implemented their framework on top of Giraph, an open-source implementation of the Pregel model.

Besides our work, the only other proposal addressing relational MT distribution is Lintra, by Burgueño et al. [22], based on the Linda coordination language. Lintra uses the master-slave design pattern, where slaves are in charge of executing in parallel the transformation of sub-models of the input model. The same authors propose a minimal set of primitives to specify distributed model transformations, LintraP [23]. With respect to our approach, Lintra requires to explicitly use distribution primitives, but it can be used in principle to distribute any transformation language by compilation.

2.1.2. Shared-memory parallelization for graph processing languages

Shared-memory parallelization is a closely related problem to distribution. For model transformation, Tisi et al. [24] present a systematic two-steps approach to parallelize ATL transformations. The authors provide a multi-threaded implementation of the ATL engine, where each rule is executed in a separate thread for both steps. The parallel ATL compiler and virtual machine have been adapted to enable a parallel execution and reduce synchronization overhead.

A similar approach for parallel graph transformations in multi-core systems [25] introduces a two-phase algorithm (matching and modifier) similar to ours. Bergmann et al. propose an approach to parallelizing graph transformations based on incremental pattern matching [26]. This approach uses a message passing mechanism to notify of model changes. The incremental pattern matcher is split into different containers, each one is responsible for a set of patterns. The lack of distributed memory concerns makes these solutions difficult to adapt to the distributed computing scenario. Moreover, in these cases, the authors investigate task distribution, while we focus on data distribution, especially for handling VLMs.

2.1.3. Alternative execution semantics for scalable transformations

The idea of overloading the execution semantics of model transformation languages in order to improve scalability is not new. In particular, several works introduced incremental or streaming computation semantics.

Incremental computing is a software feature aiming at saving the program re-computation time every time a piece of data changes. It identifies the outputs which depend on the changed data, then updates their value. EMF-IncQuery [27, 28] is a declarative model query framework for EMF models using the graph pattern formalism as a query specification language. It aims at bringing the benefits of graph pattern-based queries and incremental pattern matching to the EMF ecosystem. Bergmann et al. [29] proposed an approach to integrate incremental pattern matching into existing legacy systems built over RDBs. It translates graph patterns/rules into SQL triggers. Additional tables are added to store cached relationships. Jouault et al. [30] introduced an approach for incremental transformations in ATL. Neither ATL's syntax nor its semantics sustained any change, except for some minimal changes to the compiler. Giese et al. [31] proposed an approach to automatically induce automatic incremental synchronization using Triple Graph Grammar (TGG).

Streaming computation is a computer programming paradigm equivalent to event stream processing and reactive programming. Several approaches that support reactive and streaming model transformation have been proposed [32, 33, 34]. VIATRA 3 [32] is a source incremental event-driven model transformation platform based on the reactive programming paradigm. VIATRA 3 offers a family of internal DSLs to specify advanced tool features built on top of existing languages like EMF-IncQuery and Xtend. VIATRA-CEP [33] is an engine for streaming model transformations by combining incremental transformation and complex event processing. It includes a DSL for defining atomic event classes and combining them into complex patterns and events. Martínez et al. [34] introduced a reactive model transformation engine for ATL. This work combines efforts on enabling incrementality and lazy evaluation of ATL transformations. These approaches are well-suited to the model-driven applications involving frequent runtime updates. The transformation engine takes care of re-executing only the necessary computation affected by the update. For efficient execution of one-shot transformation on VLMs, these approaches fail drastically.

2.1.4. High-level languages for distributed data-parallel computing

Many high-level languages for data-parallel computing targeting distributed programming models have been proposed. However, these languages are not designed for performing distributed model transformations.

Microsoft SCOPE [35], Pig Latin [36], and HiveQL [37] are high-level SQL-like scripting languages targeting massive data analysis on top of MapReduce. Pig Latin and SCOPE are hybrid languages combining both forces of a SQL-like declarative style and a procedural programming style using MapReduce primitives. They provide an extensive support for user-defined functions. Hive is a data warehousing solution built on top of Hadoop. It comes with a SQL-like language, HiveQL, which supports data definition statements to create tables with specific serialization formats, and partitioning and bucketing columns.

DryadLINQ is a language designed to target the Dryad [38] engine, a general-purpose distributed execution engine for coarse-grain data-parallel applications. Unlike PigLatin or SCOPE, which introduce new domain-specific languages, DryadLINQ is embedded as constructs within existing programming languages. A program written in DryadLINQ is a sequential program composed of expressions specified in LINQ, the **.NET** Language Integrated Query. The expressions perform arbitrary side-effect-free operations on datasets and can be debugged using standard **.NET** development tools.

2.2. Scalable persistence of VLMs

The interest on scalable model persistence has grown significantly in recent years. Nonetheless, existing approaches are still not suitable to manage this kind of artifacts both in terms of processing and performance.

2.2.1. XMI-based approaches

Models stored in XMI need to be fully loaded in memory for persistence. The lack of support for lazy or partial loading of models hampers handling VLMs

not fitting in a memory of a single machine. Moreover, this persistence format is not adapted to developing distributed MDE-based tools. One way to tackle scalability issues while sticking to the XMI representation is by decomposing the model into fragments. Amálio et al. [39] proposed a mathematical ground for this, based on the ideas of modularity and separation of concerns. Below, we investigate the state-of-the-art tools and frameworks for persisting EMF-based models and draw down their limitations.

EMF fragments [40] is a hybrid persistence layer for EMF models aimed at achieving fast storage and navigation of persisted data. EMF-Fragments uses annotations to decompose a model into smaller documents. A similar approach is EMF Splitter [41], it borrows the modularity structure used in Eclipse for Java projects organization to decompose the models. Both approaches rely on the proxy resolution mechanism used by EMF for inter-document relationships. EMF Fragment supports MongoDB, Apache HBase, and regular files, while EMF Splitter supports only XMI files.

2.2.2. Relational-based approaches

Connected Data Objects (CDO) model repository [42] is the *de facto* standard solution to handle large models in EMF by storing them in a relational database. It was initially envisioned as a framework to manage large models in a collaborative environment with a low memory footprint. However, different experiences have shown that CDO does not scale well to very large models [43, 44, 45]. CDO implements a client-server architecture with transactional and notification facilities where model elements are loaded on demand. CDO servers (usually called repositories) are built on top of different data storage solutions. However, in practice, only relational databases are commonly used.

2.2.3. NoSQL-based approaches

Barmpis and Kolovos [46] suggest that NoSQL databases would provide better scalability and performance than relational databases due to the interconnected nature of models. Morsa [44] was one of the first approaches to provide persistence of large-scale EMF models using NoSQL databases. Specifically, Morsa uses MongoDB [47], a document-oriented database, as its persistence backend. Morsa can be used seamlessly to persist large models using the standard EMF mechanisms. As CDO, Morsa is built using a client-server architecture. Morsa provides on-demand loading capabilities together with incremental updates to maintain a low workload. The performance of the storage backend and their own query language (MorsaQL) have been reported in [44] and [45].

Mongo EMF [48] is another alternative to storing EMF models in MongoDB. Mongo EMF provides the same standard API as previous approaches. However, according to the documentation, the storage mechanism behaves slightly different than the standard persistence backend (for example, when persisting collections of objects or saving bi-directional cross-document containment references). Using Mongo EMF to replace another backend in an existing system, requires an adaptation process.

Table 1: Summary of distributed and parallel MTs approaches

Approach	Paradigm		Exec. Mode		Concurrency		EMF Integ.
	Graph	Rel.	Shared	Dist.	Mem.	Disk	
VMTS-para	●	○	●	○	●	○	○
VMTS-dist	●	○	○	●	○	●	○
ATL-para	○	●	●	○	●	○	●
LinTra	○	●	●	●	●	○	○
Henshin	●	○	○	●	○	●	○
Tung et al.	●	○	○	●	○	●	○

Legend: ● feature supported, ○ feature not supported

Table 2: Summary of model persistence approaches

Approach	Store	Lazy loading	Access	Conc. write
CDO	SQL	fine	centralized	yes
Mongo EMF	NoSQL	fine	centralized	yes
Morsa	NoSQL	fine	local	no
EMFSplit	XMI	coarse	local	diff. splits*
EMFfragments	XMI	coarse	decentralized	diff. splits

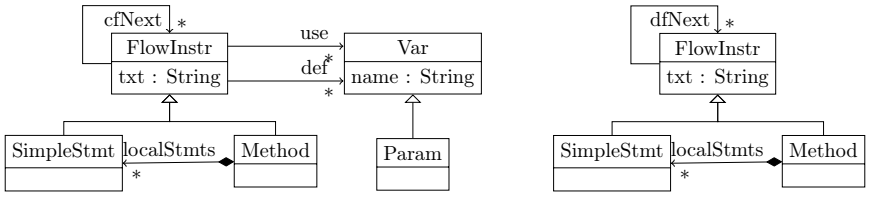
(*)Concurrent writes are not allowed at the file level. Permitted only on different files.

2.3. Current limitations to scalable MT and persistence

In this section, we reviewed existing approaches aiming at optimizing graph processing, with a particular focus on graph transformation operations. Moreover, we presented state-of-the-art tools for the persistence of VLMs in EMF. Tables 1 and 2 summarize the limitations of MT approaches and model persistence tools respectively.

While most approaches in graph/model transformation focus on the parallelization of the transformation using the graph paradigm, only two approaches focus on relational model transformation, *parallel-ATL* [24] and *LinTra* [22]. Nonetheless, parallel-ATL is still limited to the resources of one machine, while LinTra is not compatible with existing EMF-based applications, especially precluding some runtime interactions with EMF components. It translates the transformation rules together with in-/output metamodels to a Linda-specific format. As for the remaining approaches, visibly, none of them has a clear support of concurrent write on target models, nor a clear support for complex transformation rules (e.g. rules involving multiple input patterns). Our solution is designed to address these missing features.

Concerning the solutions for persisting EMF models, we observed that, while most of the existing approaches scale in some scenarios, they expose a few limitations that are crucial to distributed model Transformations. *EMF Fragments* and *EMF Splitter* require a good fragmentation from the user. They support lazy loading only at the chunk level. As a consequence, complex queries may lead to loading the whole model even though some elements are not accessed. Moreover, EMF Fragments on top of HBase is not transparent with regard to



(a) ControlFlow metamodel excerpt (b) DataFlow metamodel excerpt

Figure 1: Simplified ControlFlow and DataFlow metamodels

model distribution. Queries and transformations need to explicitly take into account that they are running on a part of the model and not the whole model. These backends assume to split the model into balanced chunks. This may not be suitable for distributed processing, where the optimization of computation distribution may require uneven data distribution. Finally, existing solutions using a client-server architecture (e.g. CDO over a distributed database) use a single access point. Even when model elements are stored in different nodes, access to model elements is centralized, since elements are requested from and provided by a central server. This constitutes a bottleneck and does not exploit a possible alignment between data distribution and computation distribution. With a growing size of clients, the single access point can rapidly turn into a limitation.

We extended an existing multi-layer persistence backend for VLMs in EMF with support for concurrent read-write. The distribution is completely transparent for EMF-based tools, and the alignment between data distribution and computation distribution is alleviated. Moreover, our solution relies on the CAP (Consistency, Availability and Partitioning tolerance) principle, where we sacrifice some ACID properties in order to achieve better global performance while guaranteeing full consistency.

3. Background

Before going into the details of our proposal, in this section, we present the necessary background. First, we introduce the ATL transformation language by means of a case study. We discuss a set of properties that reduces inter-rules communication. Later, we introduce the MapReduce programming model and describe its properties and features.

3.1. The ATL transformation language

While our distribution approach is applicable to the class of relational model transformation languages, in this paper, we refer to the ATL language to exemplify this class. To elucidate the discussion of our approach, we refer throughout the paper to a single case study related to the analysis of dataflows in Java programs. The case study is well-known in the MDE community, being proposed

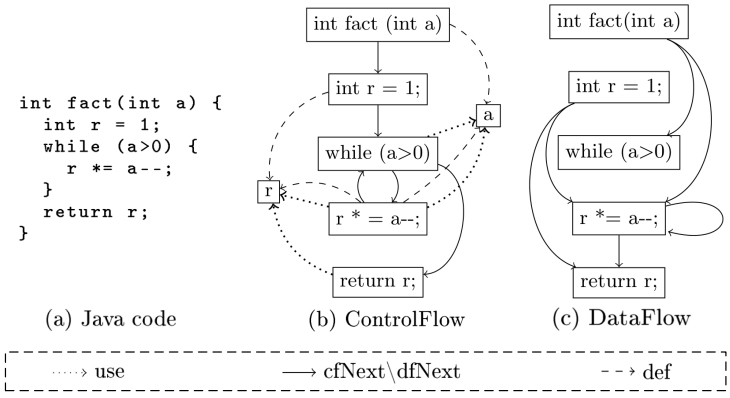


Figure 2: ControlFlow2DataFlow transformation example

by the Transformation Tool Contest (TTC) 2013 [49] as a benchmark for MT engines.

Excerpts of the source and target metamodels of this step are shown in Fig. 1. In a control-flow diagram (Fig. 1a), a *FlowInstruction* (FlowInstr) has a field *txt* containing the textual code of the instruction, a set of variables it defines or writes (*def*), and a set of variables it reads (*use*). A method may contain a set of simple statements *localStmts*. A *FlowInstruction* points to the potential set of instructions that may be executed after it (*cfNext*). *Method* signatures and *SimpleStatements* (SimpleStmt) are kinds of *FlowInstruction*. A *Parameter* is a kind of *Variable* that is defined in method signatures.

The dataflow diagram (Fig. 1b) has analogous concepts of *FlowInstruction*, *Method* and *SimpleStatements* but a different topology based on the dataflow links among instructions (*dfNext*). For every flow instruction n, a *dfNext* link has to be created from all nearest control-flow predecessors m that define a variable which is used by n.

Fig. 2 shows an example of models for each metamodel, derived from a small program calculating a number factorial. For readability reasons, and in order not to congest our graphs, containment references are omitted. As it can be seen in the figure, the transformation changes the topology of the model graph, the number of nodes and their content, and therefore can be regarded as a representative example of general transformations. In this paper we refer to an ATL implementation of the transformation named *ControlFlow2DataFlow* and available at the tool website[2].

Languages like ATL are structured in a set of transformation rules encapsulated in a transformation unit. These transformation units are called *modules* (Listing 1, line 1). The query language used in ATL is the OMG's Object

[2] `https://github.com/atlanmod/ATL_MR/`

Constraints Language (OCL) [50]. A significant subset of OCL data types and operations is supported in ATL. Listing 1 shows a subset of the rules in the *ControlFlow2DataFlow* transformation.

Input patterns are fired automatically when an instance of the source pattern (a match) is identified, and produce an instance of the corresponding target pattern in the output model. Implicitly, transient tracing information is built to associate input elements to their correspondences in the target model.

Source patterns are defined as OCL *guards* over a set of typed elements, i.e. only combinations of input elements satisfying that guard are matched. In ATL, a source pattern lays within the body of the clause '*from*' (Listing 1, line 15). For instance, in the rule *SimpleStmt*, the source pattern (Listing 1, line 16) matches an element of type *SimpleStmt* that defines or uses at least a variable. Output patterns, delimited by the clause '*to*' (Listing 1, line 18) describe how to compute the model elements to produce when the rule is fired, starting from the values of the matched elements. E.g., the *SimpleStmt* rule produces a single element of type *SimpleStmt*. A set of OCL *bindings* specify how to fill each of the features (attributes and references) of the produced elements. The binding at line 20 copies the textual representation of the instruction, the binding at line 21 fills the *dfNext* link with values computed by the *computeNextDataFlows* OCL *helper*. The rule for transforming methods is similar (Listing 1, lines 3-12).

ATL matched rules are executed in two phases, a *match phase* and an *apply phase*. In the first phase, the rules are applied to source models' elements satisfying their guards. This execution strategy is recognized in the community as Map Entities before Relations/Objects before Links model transformation design pattern [51]. Each single match corresponds to the creation of an explicit traceability link. This link connects three items: the rule that triggered the

Listing 1: ControlFlow2DataFlow - ATL transformation rules (excerpt)

```
1  module ControlFlow2DataFlow;
2  create OUT : DataFlow from IN : ControlFlow;
3  rule Method  {
4    from
5      s : ControlFlow!Method
6    to
7      t : DataFlow!Method (
8        txt <- s.txt,
9        localStmts <- s.localStmts,
10       dfNext <- s.computeNextDataFlows()
11     )
12 }
13
14 rule SimpleStmt {
15   from
16     s : ControlFlow!SimpleStmt (not(s.def->
17           isEmpty() and s.use->isEmpty()))
18   to
19     t : DataFlow!SimpleStmt (
20       txt <- s.txt,
21       dfNext <- s.computeNextDataFlows()
22     )
23 }
```

application, the match, and the newly created output elements (according to the target pattern). At this stage, only output pattern elements type is considered, bindings evaluation is left to the next phase.

The apply phase deals with the initialization of output elements' features. Every feature is associated to a binding in an output pattern element of a given rule application. Indeed, a rule application corresponds to a trace link. Features initialization is performed in two steps, first, the corresponding binding expression is computed. Resulting in a collection of elements, it is then passed to a resolution algorithm (called *resolve algorithm*) for final update into the output model. The *resolve algorithm* behaves differently according to the type of each element. If the type is primitive (in case of attributes) or target, then it is directly assigned to the feature. Otherwise, if it is a source element type, it is first resolved to its respective target element – using the tracing information – before being assigned to the feature. Thanks to this algorithm we are able to initialize the target features without needing to navigate the target models. The `resolveTemp`, a generalization of the *resolve algorithm*, is also invoked on a source element but returns a specific target model element identified by a name of the rule and a name of a target pattern belonging to this rule. A normal `resolve` can be regarded as a `resolveTemp` call having as parameters a default rule name and the name of the first target pattern element.

As result of ATL's execution semantics, especially four specific properties of the language (below), inter-rule communication is made discernible and the odds of running into race conditions are minimized. More precisely, interaction among ATL transformation rules are reduced to bindings resolution, where a target element's feature needs to link to other target elements created by other rules:

Property 1. *`Locality:` Each ATL rule is the only one responsible for the computation of the elements it creates, i.e., the rule that creates the element is also responsible for initializing its features. In the case of bidirectional references, responsibility is shared among the rules that create the source and the target ends of the reference.*

Note that if a transformation language does not satisfy this property, a way to lower the data communication cost would be by making sure that different rules sharing update task reside on the same machine.

Property 2. *`Single assignment on target properties:` The assignment of a single-valued property in a target model element happens only once in the transformation execution. Multi-valued properties can be updated only by adding values but never deleting them.*

If a language does not guarantee this property, one way of communicating less data is by local aggregating operations. Let's take for instance the example of a rule that, for every rule application increments a variable, instead of sending a bunch of increment values, it would be recommended to aggregate them and send only a single value that sums up all the increment operations.

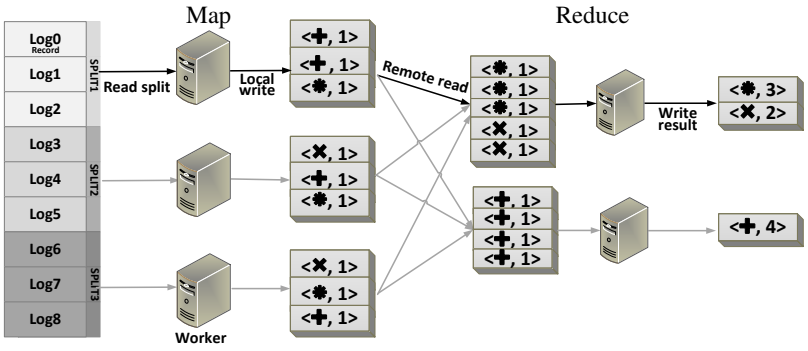


Figure 3: MapReduce programming model overview

Property 3. *`Non-recursive rule application:` Model elements that are produced by ATL rules are not subject to further matches. As a consequence, new model elements cannot be created as intermediate data to support the computation.*

This differentiates ATL from typically recursive graph-transformation languages. The property should not be confused with recursion in OCL helpers that are responsible for intermediate computations over the source models only but not the target ones.

Property 4. *`Forbidden target navigation:` Rules are not allowed to navigate the part of the target model that has already been produced, to avoid assumptions on the rule execution order.*

This property is possible thanks to the resolve algorithm. A way to workaround the non-satisfaction of this property is by making sure that the target elements creation and update happen in two different phases.

3.2. MapReduce

MapReduce is a programming model and software framework developed at Google in 2004 [11]. It allows easy and transparent distributed processing of big data sets while concealing the complex distribution details a developer might cross. MapReduce is inspired by the map and reduce primitives that exist in functional languages. Both *Map* and *Reduce* invocations are distributed across cluster nodes, thanks to the *Master* that orchestrates jobs assignment.

Input data is partitioned into a set of chunks called *Splits* as illustrated in Fig. 3. The partitioning might be monitored by the user through a set of parameters. If not, *splits* are automatically and evenly partitioned. Every *split* comprises a set of logical *Records*, each containing a pair of ⟨key, value⟩.

Given the number of *Splits* and idle nodes, the Master node decides the number of workers (slave machines) for the assignment of *Map* jobs. Each *Map worker* reads one or many *Splits*, iterates over the *Records*, processes the ⟨key, value⟩ pairs and stores locally the intermediate ⟨key, value⟩ pairs. In the

meanwhile, the *Master* receives periodically the location of these pairs. When *Map workers* finish, the *Master* forwards these locations to the *Reduce workers* that sort them so that all occurrences of the same key are grouped together. The *mapper* then passes the key and list of values to the user-defined reduce function. Following the reduce tasks achievement, an output result is generated per reduce task. Output results do not need to be always combined, especially if they will subsequently be processed by other distributed applications.

Let's take a closer look at the MapReduce programming model by means of a simple example, depicted in Fig. 3. Assume we have set of log entries coming from a git repository. Each entry contains information about actions performed over a particular file (creation $\rightarrow +$, deletion $\rightarrow X$, or modification $\rightarrow *$). We want to know how many times each action was performed, using MapReduce. The master evenly splits the entries among workers. For every record (log entry), the map worker extracts the action type and creates a ⟨key,value⟩ pair with a key the action itself and value '1'. In the reduce phase, pairs with the same key are grouped together. In our example, the modification and deletion go to the first reducer, while the creation goes to the second one. For each group, the reducer combines the pairs and creates a ⟨key,value⟩ pair, but this time with value the sum of the values with the same key. This value refers to how many times the action occurred in the logs.

A useful optimization feature shipped with MapReduce is the Combiner. It is an optional class taking place right after the map phase, and before the shuffle phase. It operates on pairs originating from the mapper running on the same node. For each set of pairs sharing the same key, values are aggregated in order to combine them into a single pair according to the *Combiner* class definition. As you can notice, the main benefit using combiners is to reduce the volume of data sent between the Map and Reduce phases, and therefore the time that is taken to shuffle different pairs across the cluster. We use the combiner feature in order to perform a local resolve, which allows us to send fewer traces to the reduce phase.

It is to the Distributed File System (DFS) that the MapReduce framework owes its ability to scale to hundreds of machines in a cluster. The master node in MapReduce tends to assign workloads to servers where data to be processed is stored to maximize data locality. An interest of MapReduce is due to its fault-tolerant processing. The *Master* keeps track of the evolution of every worker execution. If after a certain amount of time a worker does not react, it is considered as idle and the job is re-assigned to another worker. Same for DFS, data is divided into blocks, and copies of these blocks are stored in different nodes across the cluster to achieve a good availability as nodes fail.

Hadoop [52] and Hadoop Distributed File System (HDFS) [53] are the most well-known and widely used open-source implementations of MapReduce and Google-DFS respectively.

4. Conceptual Framework

In this section, we give a global overview of our distributed MT framework. We first introduce some definitions and properties we believe would help better grasp our data distribution approach. Later, we describe our distributed model transformation process by means of a simple example.

4.1. Definitions

In typical relational MT engines (e.g., the standard ATL and ETL engines), the transformation execution starts by loading the input model. Then the engine applies the transformation by selecting each rule, looking for matches corresponding to the input pattern, and finally generating the appropriate output elements [54]. Each execution of a matched input pattern is called a rule application.

From now on, we denote by $\mathcal{E}$ a set of model elements, and $\mathcal{M}$ a set of commodity machines in a distributed system $\mathcal{S}$.

Definition 1. *Let $\mathcal{R}$ be a set of model transformation rules. A rule application is defined by the tuple (r, in) where:*

- *$r \in \mathcal{R}$ is the rule being applied*
- *$in \subseteq \mathcal{E}$ is the list of source elements matched by the rule r (i.e., the input pattern)*

In the context of distributed model transformations we define other two properties of rule applications:

- *$e \in in$ (primary trigger) is a single element elected as the primary trigger of the rule application*
- *$ctx \subseteq \mathcal{E}$ (context) is the subset of model elements accessed to evaluate expressions in r for the rule application. These elements include the elements of the input pattern*

Given Definition 1, we consider a MT execution job as the union of elementary rule application execution jobs, where each job is responsible for transforming a single input pattern. In the case of rules with n-ary input pattern (matching a sub-graph), we consider the job of applying the rule to be primarily triggered by one input pattern element. Selecting a primary triggering element for each rule application ensures that, after distributing the source model, a rule application occurs on only one machine, i.e. the one responsible for transforming the triggering element.

The distribution of a transformation based on a data-parallel distribution approach over m machines ($m = |\mathcal{M}|$), consists of dividing the input model into m splits and assigning disjoint sub-models of the input model to different machines. Each machine will be then responsible for transforming the assigned subset. In what follows we refer to this set of elements assigned to a machine i by $\mathcal{A}_i$. Given a system $\mathcal{S}$ of m machines, the set of assigned elements has the following property:

Property 5. *Each element $e \in \mathcal{A}_i$ is assigned to one and only one set ($\forall i, j \in \mathcal{M},\ i \neq j \implies \mathcal{A}_i \cap \mathcal{A}_j = \emptyset$)*

In order to transform its set of assigned elements $\mathcal{A}_i$, a machine i needs to access all the elements that are necessary for the transformation of its assigned elements. We denote this set as:

$$\mathcal{D}_i = \bigcup_{e \in \mathcal{A}_i} dependencies(e)$$

where $dependencies(e)$ is the union of the contexts of all the rule applications that are primarily triggered by e.

Consequently, every machine i needs to load all the elements $\mathcal{L}_i$ belonging to $\mathcal{A}_i \cup \mathcal{D}_i$.

Typical distributed graph processing presents a higher ratio of data access to computation w.r.t. typical scientific computing applications. In particular, most of the computational complexity of MT rules lies in the pattern matching step, i.e. the exploration of the graph structure. With a naive data distribution scheme, the execution time can be monopolized by the wait time for model elements lookup. Hence, an intelligent assignment of elements to cluster nodes should be performed. For instance, an intelligent data-distribution strategy may try to minimize network traffic by an intelligent assignment ($\mathcal{A}_i$). In particular, it may try to minimize the number of shared elements, i.e. elements that are needed by multiple nodes (e.g. i and j share $\mathcal{L}_i \cap \mathcal{L}_j$). The problem is similar to a well-known problem in the graphs community, *Graph-Based Data Clustering with Overlaps* [55]. This problem allows clusters overlapping by duplicating (to a given extent) graph vertices or edges. In related work [56], we discuss this problem in more details. We have formalized the problem of model partitioning for distributed model transformations in linear programming and proposed a greedy algorithm for efficient data distribution.

4.2. Overview of the distributed transformation process

Figure 4 shows a global overview of our distributed transformation framework by means of a simple example. The transformation is composed of a simple rule that changes the shape of nodes (from Square to Hexagon) but keeps the same ID as well as graph topology. Our distributed cluster is composed of a single master node, data nodes, and task nodes. Data nodes and task nodes communicate with each other through a distributed MOF-compliant model access and persistence API. While task nodes are responsible for transforming the input model or composing the output one, data nodes are responsible for hosting the partial input/output models. Although in our illustration we differentiate between data nodes and task nodes, in real-world clusters, data can be processed in the same node it resides in.

Our distributed model transformation process is divided in three steps, (i) *data distribution*, (ii) *parallel local transformation*, and (iii) *parallel composition*. The coordination phase plays the role of a barrier in which task nodes share their output data among each other for composition.

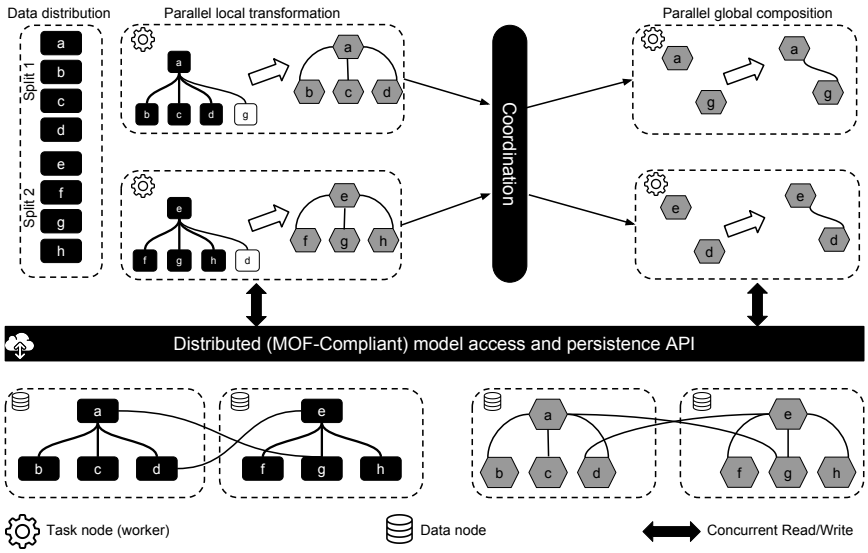


Figure 4: A conceptual framework for distributed model transformation

In the first phase, the master node is responsible for assigning source model elements to task nodes (*data distribution*) for computation. Each task node is responsible for executing the rule application triggered by the assigned elements. These subsets (a.k.a. chunks, splits, or partitions) are designed to avoid any redundant computation in two separate nodes (respecting Property 5). Moreover, since the master node is limited by its memory capacity, we consider a lightweight and memory-friendly assignment mechanism of source model elements. In our example, the master node assigns {a, b, c, d} to the upper node, and the remaining to the second one (as shown in Figure 4).

After assigning source model elements to task nodes, they start processing the transformation in parallel. However, due to the complexity of the pattern matching phase, a task node may have to load additional elements in order for its local transformation to complete (*parallel transformation*). For instance, in our example, each square needs to traverse the set of its direct neighboring nodes to reproduce the graph topology. In particular, the upper task node needs the elements "g" while the second node needs "d". Because the set of additional elements is known only at runtime, our persistence framework transparently provides task nodes with the ability to access any input element, identified by its UUID, during the transformation execution. This is granted to an on-demand lazy loading mechanism (see Section6). Additionally, all data requests are transparently passed through the persistence layer and task nodes would not be aware of the physical location of the model elements. However, the cost for accessing data is not constant as it is likely influenced by the network I/O.

At the end of the parallel transformation phase, each task node will result in a local output sub-model together with a set of tracing information that

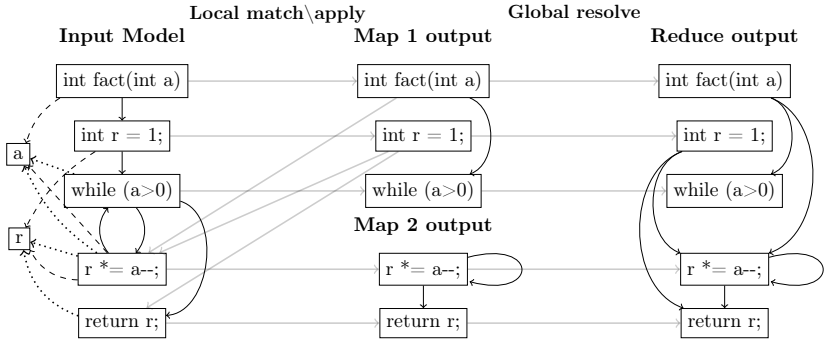


Figure 5: ControlFlow2DataFlow example on MapReduce

has a twofold role. This information does not only contain information about computation already performed but also about the one that needs to be performed in order to compose local output sub-models into a global output model.

The coordination phase plays the role of a synchronization barrier, where, once all task nodes finish their parallel transformation, they exchange trace information about the missing links in order to compose the global model. To avoid network congestion, target sub-models are stored together with the traces, and only a list of UUIDs identifying these traces is passed through the network.

In the final phase (*parallel composition*), missing tracelinks are evenly split to task nodes, and relationships are established between the output sub-models to compose the final output model. The persistence framework enables task nodes to concurrently establish links between any given model elements. Similarly to the previous phase, task nodes can transparently access, at any time, model elements and the transformation traces.

5. Distributed Relational Model Transformation on MapReduce

Distributed model-to-model transformation inherits most of the well-known challenges of efficient parallel graph processing, such as poor data locality and unbalanced computational workloads. In particular, implicit data distribution is not trivial for transformation languages where rules applied to different parts of the model can interact in complex ways with each other. The higher is the number of these interactions, the bigger is the volume of data communicated across the network, both at inter and intra-phase levels. In MapReduce, this turns into, more data to serialize, de-serialize, and less network throughput due to congestion. Thanks to ATL properties introduced in Section 3, the possible kinds of interaction among ATL rules is strongly reduced, which allows to decouple rule applications and execute them in independent execution units.

In this section, we show how our distributed transformation process has been mapped to the MapReduce programming model, then, we illustrate the alignment of ATL execution semantics to the MapReduce.

5.1. ATL and MapReduce alignment

Mostly, distributed (iterative) graph processing algorithms are data-driven, where computations occur at every node of the graph as a function of local graph structure and its intermediate states. The transformation algorithm in ATL could be regarded as an iterative graph processing algorithm with two phases (match and apply described in Section 3) and one intermediate state (matched). Each node of the cluster that is computing in parallel takes charge of transforming a part of the input model.

In our approach, we propose a mapping that aims at reducing cross-machine communication cost. That is by adopting some good practices for scalable graph processing in MapReduce. This is made possible, thanks to the alignment of the ATL distributed execution semantics with MapReduce described below. The proposed mapping is conceptually similar to the original ATL algorithm.

As an example, Figure 5 shows how the ATL transformation of our running example could be executed on top of a MapReduce architecture comprising three nodes, two maps and one reduce workers. The input model is equally split according to the number of map workers (in this case each map node takes as input half of the input model elements). In the map phase, each worker runs independently the full transformation code but applies it only to the transformation of the assigned subset of the input model. We call this phase *Local match-apply*. Afterwards, each map worker communicates the set of model elements it created to the reduce phase, together with trace information. These trace links (grey arrows in Figure 5) encode the additional information that will be needed to `resolve` the binding, i.e. identify the exact target element that has to be referenced based on the tracing information. The reduce worker is responsible for gathering partial models and trace links from the map workers, and updating properties value of unresolved bindings. We call this phase *Global resolve*.

In the following, we briefly describe the distributed execution algorithm, which is decomposed in two phases, the *Local match-apply* phase assigned to mappers and the *Global resolve* phase assigned to reducers.

Local match-apply

At the beginning of the phase, input splits are assigned to map workers. Each one of these splits contains a subset of the input model for processing. Despite this, each worker can access the whole input models in case it needs additional data for bindings computation. Thanks to our model data mapping, which adheres to good practices in graph data representation [57], model elements together with their adjacent nodes are stored as tuples. Adjacent model elements are accessed lazily and on-demand when it is required by the transformation execution logics. Algorithm 1 illustrates the pseudo-code of the *Local match-apply*.

For every model element in the split, the map function verifies if a rule guard matches and in this case instantiates the corresponding target elements, same as in the regular execution semantics. In the case of rules that match multiple

Algorithm 1: Map function

```
   input : Long key, ModelElement element
 1 foreach rule ∈ getApplicableRules(element) do
 2     if isMatched(element, rule) then
 3         link ← createLink(element, rule);
 4         addLink(linkSet, link);
 5         foreach binding ∈ getBindings(link) do
 6             if isAttribute(binding) then
 7                 applyAttribute (binding);
 8             else
 9                 foreach ComputedElement ∈ computeBindingExp(binding)
                    do
10                     if isLocal(ComputedElement) then
11                         resolvedElement ←
                            resolveTarget(ComputedElement);
12                         applyReference(resolvedElement, binding);
13                     else
14                         addElementToTrace(ComputedElement, binding);
15         storeLink(generatedKey, link);
           // generatedKey to decide to which reducer this link will be
              assigned
```

elements, the map function would consider the elements of the split as the first element of the matched pattern, and look for combinations of other elements satisfying the guard. For each instantiated output element, a trace link is created connecting source and target elements of the applied rule. Subsequently, the algorithm starts processing the list of property bindings for the instantiated target elements. We extended the behavior of the `resolve algorithm` to enable handling elements transformed in other nodes, we call this algorithm `local resolve`. In the case of attribute bindings (lines 6–7), the same standard behavior is preserved, the OCL expression is computed and the corresponding feature is updated accordingly. While bindings related to references connect elements transformed by different rule applications, potentially on different nodes, the resolution is performed in two steps: (i) the OCL expression of the binding computes to a set of `elements in the source model` and ATL connects the bound feature to these source elements using trace links; (ii) the source-models elements are resolved, i.e. substituted with the corresponding target element according to the rule application trace links. If the source and target elements of the reference are both being transformed in the same node, both steps happen locally (lines 10–12), otherwise trace links are stored and communicated to the reducer, postponing the resolution step to the `Global resolve` phase. `The resolveTemp algorithm` behaves similarly. In the Global

resolve phase, unresolved elements coming from the `resolveTemp` are treated equally to unresolved model elements coming from the normal `resolve`.

Global resolve

At the beginning of the reduce phase, all the target elements are created, the local bindings are populated, and the unresolved bindings are referring to the source elements to be resolved. This information is kept consistent in the tracing information formerly computed and communicated by the mappers. Then it `resolves` the remaining reference bindings by iterating over the trace links. For each trace link, the reducer iterates over the unresolved elements of its property traces, resolves their corresponding element in the output model, and updates the target element with the final references. The pseudo-code for the *Global resolve* is given in Algorithm 2.

Algorithm 2: Reduce function

input : String key, Set<TraceLink> links

```
1 foreach link ∈ links do
2     foreach property ∈ getTraceProperties(link) do // unresolved
        properties
3
4         foreach elmt ∈ getSourceElements(property) do
5             resolvedElement ← resolveTarget(elmt);
6             updateUnresolvedElement(property, trgElmt);
              // equivalent to applyReference (element, binding) in map
                 function
```

5.2. ACID properties for atomic model manipulation operations in ATL

Distributed computation models like MapReduce are often associated with persistence layers for accessing distributed data. In solutions for Big Data, where scalability is the main requirement, persistence layers sacrifice ACID properties in order to achieve better global performance. In networked shared-data systems, this is also known as CAP theorem [58]. Such properties need to be implemented by the application layer on a need basis.

Hereafter, we reason about the execution semantics of ATL, especially, atomic model manipulation operations (as coming from the MOF specification). Our objective is to extract the minimal set of ACID properties to be guaranteed for a *consistent* output model. This reasoning is conducted while taking into consideration the model access modes and the possible interactions between model manipulation operations.

As no definition of a transaction has been carried in the formal MOF specification, in our study, we consider that *each model manipulation operation on a single element runs in a separate transaction.* This definition is similar to the one adopted by the CAP theory. In our analysis, we rely on the semantics of the

MOF reflection API, especially the methods used by the ATL engine. Hereafter, we give a brief description of the MOF's Reflection API.

Table 3: Access modes in ATL transformations

	MATCH	APPLY
Input	READ-ONLY	
Output	WRITE-ONLY	
Trace	WRITE-ONLY	READ-ONLY

In model-to-model transformations with ATL, models are never accessed in read-write mode but only in read-only mode or write-only mode. Precisely, while source models are always read-only and target models are always write-only, trace models are, depending on the phase, either read-only (apply phase) or write-only (match phase). Table 3 summarizes different access modes per model-kind\phase.

Moreover, in our distributed transformation framework, we identify two different scopes of computation, depending on the phase. The computation can run either in a local scope (*Local match/apply*), or a global scope (*Global resolve*). In the local scope, all the write operations happen in local. Data is always created and manipulated by the same task node. As a result, model manipulation operations on the same model element never run concurrently. On the other hand, concurrency may take place in the global scope, and different machines can modify the same element at the same time.

Regardless of the computation scope, models that are accessed in read-only mode are subject only to side-effect free queries[3]. Likewise, during the local scope, trace models are always consistent because they are considered as local intermediate data. In the remaining of this section, we are interested only in the global scope, in particular, the output models.

Output models undergo a building process creating a model that grows monotonically (Property 2 and 3). However, during the global computation, there exist some specific cases that may lead to inconsistency. In particular, (i) when having operation invoking a change in more than one element (e.g. containment and bidirectional references), or (ii) updating multi-valued references. For instance, in ATL, elements are newly created and attached to the resource[4] (in the match phase), they're all considered as root elements. Moreover, they're linked to it through a containment-like relationship. On these terms, and despite Property 4, when moving these elements to their final container (in the apply phase), first, they are removed from the resource, then, they are moved to the new containing element. Similarly, in the case of updating a containment or a bidirectional reference. These operations need to either entirely fail or succeed.

[3]The accessed data is always consistent since no write operation occurs

[4]A resource can be considered as an index for root elements

Table 4: Summary of accesses counts of MOF Reflection operations

Methods	Max count*		Properties
	Read	Write	
Operations on properties			
`get*`	1	0	_C_D
`set*`	2	2	AC_D
`isSet*`	1	0	_C_D
`unset*`	0	1	_C_D
Operations on multi-valued properties			
`add`	2	2	AC(I)D
`remove`	2	2	AC(I)D
`clear`	1	0	_C_D
`size`	1	0	_C_D

* Note that only max access count is taken under consideration

Table 4 depicts the set of operations in the MOF Reflection API as well as the maximum read/write operation count within a transaction. Note that, operations on full objects such as elements creation are not considered. Finally, the table depicts the ACID properties that need to be fulfilled when the operation is performed during model transformation, in order to guarantee the correctness of the output model. The properties between parenthesis can be relaxed[5], while the others should be strongly preserved.

Beyond lowering data communication over the network, the interest of ATL properties, discussed in this section, extends to reducing the chances of running into concurrent modifications, and thus the *Isolation* property can be relaxed for some operations. Especially, thanks to Property 2, updates on single-valued properties occur only once in the transformation lifetime. In the case of updates on multi-valued properties, the chances of having concurrent updates are definitely considerable. Nevertheless, in ATL only *add* or *remove* operations might run into concurrent updates. The distributed persistence framework should make sure that two concurrent *add* or *remove* operations will leave the system in a consistent state[6].

As noticed in the *properties* column, `Durability` and `Consistency` are two mandatory properties to be preserved. The correctness of the output results is tied to guaranteeing these properties in every single method. Supposing that the underlying backend guarantees ACID properties at the finest-grained level (single CRUD operation), methods involving updates in more than one element (two-phased commits) need to be atomic, while update operations on multi-valued properties (`add` and `remove`) need to be isolated. These methods should execute in two steps, first, the latest value is looked-up, then the property's value is updated according to the method behavior. However, thanks to the monotonic building process of models in ATL, even if a two-phase commit does not happen

[5]Supported only in specific scenarios

[6]It is worth to mention that in MOF, only type, cardinality, and default properties are natively checked for Consistency. Model constraints, described as OCL invariants, are validated only when invoked.

in the same transaction, the model will eventually be consistent (i.e. we will end up having the same output model). Relaxable properties are depicted between parenthesis. In case one of these methods fails, the system should rollback. In the next section, we show how we guarantee the set of ACID properties on top of NeoEMF/Column.

6. Decentralized Model Persistence for Distributed MTs

In the previous section, we discussed the set of ACID properties that is needed to guarantee a sound and consistent MT in a distributed manner. In this section, we introduce a solution that realizes the aforementioned requirement on top of a decentralized wide-column store.

Storing models in the Cloud is a good solution to break down the complexity in handling VLMs. Due to the limitations discussed in Section 2, we extend, NeoEMF, a multi-layer persistence framework with a decentralized persistence layer called NeoEMF/Column. NeoEMF is EMF-compliant. EMF-based tools would not be aware of the NeoEMF framework, as communications between the tool and NeoEMF are always passing through the EMF layer. We benefit from the built-in features in NeoEMF to alleviate the performance of accessing and storing models for distributed model transformations. In particular, we rely on the on-demand loading mechanism to ensure that only needed elements are loaded, especially when the model is too big to fit in memory. Also, we exploit the different caching mechanisms shipped within NeoEMF reduce accesses to the database and improve the access time of already loaded elements.

NeoEMF/Column is built on top of HBase, a decentralized wide-column store. NeoEMF/Column hides the model distribution from client's applications. Model access to remote model elements in NeoEMF/Column is decentralized, which avoids the bottleneck of a single access point, and alleviates the alignment between data distribution and computation distribution. NeoEMF/Column is delivered with two stores, the first one is `Direct-write` and the second one is `Read-only`. The first store optimizes memory usage by reflecting model changes directly to the underlying backend. And thus, make the new changes directly available to other clients. Inconveniently, all clients have to fetch properties values from the underlying backend at every read operation. In future work, we plan to supply NeoEMF/Column with a distributed notification mechanism to alert clients of changes in model elements.

Model data in NeoEMF/Column is stored using adjacency list. For each node in the graph, a list of vertices adjacent to it is stored. Each row in the table is responsible for storing a model element. This representation has a low storage cost on disk, and the time to access the list of adjacent nodes is constant. In our design, we take advantage of the use of UUID design principle to flatten the graph structure into a set of key-value mappings. More details about the model data mapping can be found in the NeoEMF/Column tool paper [16].

Table 5: Summary of accesses counts to the underlying column-based storage system

Methods	Max count*		Rows	Properties	
	Get()	Put()**		ATL	NeoEMF
Operations on properties					
`get*`	1	0	1	_C_D	ACID
`set*`	3	2	2	AC_D	AC(I)D
`isSet*`	1	0	1	_C_D	ACID
`unset*`	0	1	1	_C_D	ACID
Operations on multi-valued properties					
`add`	3	2	2	AC(I)D	AC(I)D
`remove`	3	2	2	AC(I)D	AC(I)D
`clear`	1	0	1	_C_D	ACID
`size`	1	0	1	_C_D	ACID

* Note that only max access count is taken under consideration, likewise for rows
** Updating two cells in the same row counts for one single `Put`

6.1. Guaranteeing ACID properties for distributed MTs with ATL

HBase is not a strongly ACID-compliant database, it provides ACID semantics only on a per-row basis. It is up to users to employ this semantics to adapt ACID properties according to the behavior of their applications. Notably, HBase is shipped with the following warranties:

- Any `Put` operation on a given row either entirely succeeds or fails to its previous state. This holds even across multiple column families. Same for the `Append` operation
- The `CheckAndPut` is atomic and updates are executed only if the condition is met, similar to the CAS mechanism
- Any `Get` or `Scan` operation returns a complete row existing at the same point in the table's history
- `Get` operations do not require any locks. All reads while a write in progress will be seeing the previous state

In this perspective, we adapted ACID properties semantics as provided by HBase, in order to guarantee the set of ACID properties needed by ATL. Table 5 extends Table 4 with the ACID properties that are guaranteed by NeoEMF/Column, and the number of rows involved in each update. This time, the properties between parenthesis, in the column NeoEMF/Column, refer to the properties partially supported by NeoEMF/Column.

Hereafter, we describe how we implement and guarantee the set of ACID properties needed by ATL:

Atomicity — Modifications on a single object's properties are atomic. Modifications involving changes in more than one object are not. In this case, one way to provide atomicity is by manually implementing a rollback operation that undoes the changes affected by this commit. Since commits can be composed by at most 2 `Put` calls, the cost of rolling-back is low and changes

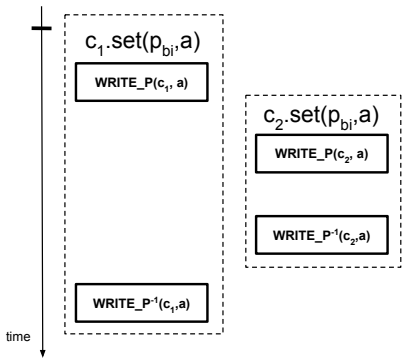


Figure 6: Fail scenario of two concurrent calls on a 1-to-1 bidirectional reference. After both updates finish, they leave the system in an inconsistent state where, $c_1 \underset{p}{\rightarrow} \emptyset$, $c_2 \underset{p}{\rightarrow} a$, and $a \underset{p}{\rightarrow} c_1$

can be tracked within the methods. In case the first `Put` succeeds but not the second, then there is only one `Put` to undo. Otherwise, no rollback is needed.

Consistency — Modifications on object's properties are always consistent. For that, we rely on EMF, which validates some consistency rules before a commit happens. The validation of rules issued by OCL invariants is not supported by EMF, likewise for NeoEMF/Column. Failures in single-valued properties are handled by HBase, while multi-valued are handled by the framework (See Atomicity and Isolation).

Isolation — There are particular cases where isolation is not guaranteed in NeoEMF/Column for the `set`, `add`, and `remove` methods (as shown in Table. 4). Figure 6 shows a fail scenario of two concurrent calls on a 1-to-1 bi-directional reference. Such a concurrent scenario would not occur in ATL as only one rule application is responsible for performing this action. In fact, concurrent operations occur only on multi-valued properties in order to perform a monotonic update, either by adding or removing an element at a specific index. In order not to have inconsistent result in absence of write-write synchronization, concurrent writes should be provided with `isolation`. To do so, two possible solutions exist. The first one is using row locking on every write operation. This solution is not encouraged in HBase[7], and the API allowing row-locking has been removed in the latest versions of HBase. The second one (what we use instead) is the CAS mechanism. In HBase, `CheckAndPut` operations are executed only when matching a specified condition on a property value. In NeoEMF/Column, we use it for removing elements from multi-valued properties. The operation first

[7] `https://issues.apache.org/jira/browse/HBASE-7315`

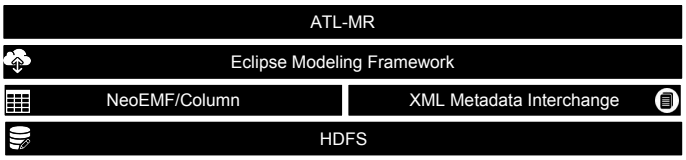


Figure 7: NeoEMF: a general view and architecture

reads the object it later plans to update and commits the changes only if the value is still as current. All of this, in one atomic isolated operation. To add elements to properties we use `Append` operation, which is also atomic and isolated. For add operations at a specific index, we infer the append offset from the index itself, since all objects' UID have the same length. It can be noted that all the case scenarios where isolation is needed in ATL are supported by NeoEMF/Column.

Durability — Modifications on a given object are always reflected in the underlying storage, even in the case of a `Data Node` failure, thanks to the replication capabilities provided by HBase.

7. Tool Support: ATL on MapReduce (ATL-MR)

Figure 7 shows the high-level architecture of our distributed engine on top of the EMF. ATL-MR runs on an Apache Hadoop [52] cluster. Hadoop is the leading open-source implementation of MapReduce. The *Master* node is responsible for distributing the input model, monitoring the ATL-MR *Slaves*, and finally returning the output results. Each node in the Hadoop cluster transparently communicates with the underlying backend through an EMF model management interface, and hence, making our distributed framework unaware of the persistence backend type. Indeed, besides supporting the de-facto serialization format, XMI [59], ATL-MR is also coupled with a multi-database model persistence framework, NeoEMF [60]. ATL-MR relies on the HDFS to distribute all of its input and output models, together with the transformation specification.

The proposed implementation adopts some of the existing good practices and design patterns for scalable graph processing in MapReduce, namely the `InMapperCombiner` and the `Shimmy` patterns proposed by Lin et al. [61]. Like normal `Combiners`, the inMapperCombiner pattern aims at reducing the number of pairs emitted by each mapper. Apart from that, this pattern runs within the mapper instead of running as a separate step. In this case, the emission of ⟨key,value⟩ pairs is held until the processing ends, then invokes the inMapperCombiner, and hence reducing the amount of data passed to the *shuffle* phase. This prevents the framework from unnecessary creation and destruction of intermediate objects.

The second pattern relates to mappers-to-reducers dataflows. It discourages diffusing the graph structure in the network. Instead, it recommends passing only the metadata of the graph elements' state, which are mostly less densely inter-connected. In our implementation, target elements are persisted, and alternatively, tracelinks are passed along to reducers. Hadoop's Map and Reduce Classes are provided with two API hooks, being `Initialize` and `Close` hooks. While the `initialize` hook is used to set-up the transformation environment, according to running phase (`Local Match-Apply` or `Global Resolve`), the `close` hook is used to execute the InMapperCombiner pattern. That is by iterating the local tracelinks (intermediate pairs) and applying a global resolve for each tracelink. This resolves the local properties and thus reduces the amount of data communicated to reducers.

ATL-MR exploits the capabilities of NeoEMF/Column, implemented on top of HBase, to distribute the storage of EMF models and enable concurrent R/W. In particular, ATL-MR uses `Direct-write` store to persist traces and target models, and uses the `Read-only` to access source models. Each supported persistent backend is shipped with specific distribution scheme. Finally, fault-tolerance in ATL-MR relies mostly on the Hadoop framework. In what follows, we describe how model distribution and access is performed in ATL-MR, then we describe how fault-tolerance is achieved.

7.1. Model distribution and access in ATL-MR

Data locality is one of the aspects to optimize in distributed computing for avoiding bottlenecks. In Hadoop, it is encouraged to run map tasks with input data residing in HDFS since Hadoop will try to assign tasks to nodes where data to be processed is stored. Each mapper is assigned a subset of model elements by the splitting process. In case of XMI models, we start first by flattening the structure of our model, which consists in producing a file containing model elements URIs as plain strings, one per line. Hadoop then takes care of shredding input data. With NeoEMF/Column, this step is not necessary since models are already stored in table-like topology.

Hadoop provides several input format classes with specific splitting behavior. Accordingly, we use different splitters depending on the persistence format of our input models. For XMI models, we use an *NLineInputFormat* on the flattened file, it takes as argument a text file and allows to specify the exact number of lines per split. Finally, the default record reader in Hadoop creates one record for each line of the input file. As a consequence, every map function in ATL-MR will be executing on a single model element. When running our distributed transformation on top of models stored in HBase, we use *TableInputFormat*. By default, this format divides at region boundaries based on a scan instance that filters only the desired cells. In our implementation, we use a *KeyOnlyfilter*. This filter accesses just the keys of each row while omitting the actual data. These keys are used later to access model elements stored in NeoEMF/Column.

For an efficient distribution of the input model, ATL-MR is shipped with two data execution modes, a greedy mode, and a random mode. While the random distribution is applicable to both of XMI and NeoEMF/hbase persistence

formats, the greedy mode is only applicable to NeoEMF/Column since, in XMI, the whole model will be loaded in any case.

Due to the file-based representation of XMI, models stored using this representation need to be fully loaded in memory, even though in practice, all the nodes need only a subset of these model elements. NeoEMF/Column, on the other hand, provides a lazy loading mechanism, which enables the ATL-MR *slaves* to transparently load only the set of needed elements. However, because of the distributed nature of NeoEMF/Column, it is mandatory to guarantee the consistency of the local values and the remote ones. For this, every time a client needs to read a value of a property, NeoEMF/Column fetches it from the underlying backend. To our convenience, we take advantage of the fact that input models are read-only[8], and we extend NeoEMF/Column with a *read-only* store. This store has the capacity to cache model elements and properties values after fetching them for the first time.

7.2. Failure management in ATL-MR

One of the two fundamental bricks of Hadoop is Yarn. It is responsible for managing resources in a Hadoop cluster. The resource manager (master) and the namenodes (slaves) altogether form the computation nodes. Namenodes are in charge of launching containers. The application master is tasked with negotiating resources and communicating with resource managers. The application startup processes as follows. First, a client submits an application to the Resource Manager, then the Resource Manager allocates a container and contacts the related Node Manager. Later the Node Manager launches the container which executes the Application Master. Finally, the application master takes care of negotiating appropriate resources and monitoring them.

In distributed applications on MapReduce, four different kinds of failure may occur, namely, *task*, *application master*, *node manager*, and *resource manager* failures. Failures can take place for too many reasons, e.g. downtime[9], runtime exceptions, etc..

Task failure. This happens when the JVM reports a runtime exception during either a map or reduce task. This exception is sent back to the application master, which marks the task attempt as failed, then frees the container to make the resource available for another task. Another scenario is when having hanging tasks. Here as well, tasks are marked as failed when the application master stops receiving heartbeats after some period of time. The timeout period is set to 10 minutes by default. When a job fails after a specific number of times, the whole Job is aborted. By default, the maximum attempts are four.

[8] We assume that input models are also not subject to changes during the transformation's execution

[9] Time during which a machine, especially a computer, is out of action or unavailable for use

Application master failure. It occurs when the resource manager stops receiving application's heartbeats. Such as tasks, application masters have also a maximum number of attempts that is set to two in the default configuration.

Node manager failure. It happens when the node crashes or runs slowly. In this case, it fails to send heartbeats periodically. The timeout period is also set to 10 minutes and all the tasks running in this node are re-executed.

Resource manager failure. Is the most serious kind of failures. It constitutes a single point of failure. If a resource manager crashes, the whole job fails and no data can be recovered. To avoid running into such a scenario, it is necessary to run a pair of resources managers in the same cluster.

In ATL-MR, in case a task solely fails (for any of the reasons above) during the map phase, then the local output model together with the intermediate traces are cleaned, and the task is relaunched. Otherwise, if a reduce task fails, then updates on the global output model are left intact. When the reduce task is run for the next time, the NeoEMF/Column assures that already serialized references are not being added to the underlying backend. In case the whole job fails, the master node cleans all the data, and the transformation is re-executed from the beginning.

8. Experimentation

In this section, we evaluate the scalability and performance of ATL-MR on top of the standard XMI backend (Sections 8.1) and our NeoEMF/Column backend (Section 8.2). Later (Section 8.3), we discuss the scenarios for which each backend is well-suited, together with their limitations.

8.1. ATL-MR on XMI

We evaluate the scalability of our proposal by comparing how the transformation of our running example performs in two different test environments. In order to be able to compare our distributed VM to the standard one, we opt, in a first place, for small models that can fit in memory. Later, we demonstrate the scalability of our approach, by scaling up to the transformation of big models. The transformation we chose has quadratic time complexity and covers a sufficient set of declarative ATL constructs enabling the specification of a large group of MTs. It also contains an interesting number of OCL operations, recursive helper's call included.

The transformation is taken from a previous case study [49] that already includes a set of input models for the benchmark. These models are reverse-engineered from a set of automatically generated Java programs, with sizes up to 12 000 lines of code. For our experiment, we used the same generation process but to stress scalability we produced larger models with sizes up to 105 000 lines of code. We consider models of these sizes sufficient for benchmarking scalability in our use case: in our experimentation, processing in a single machine the

largest of these models takes more than four hours. All the models we generated and the experimentation results are available at the tool website.

In what follows, we perform two complementary experimentations. The first one shows a quasi-linear speed-up w.r.t. the cluster size for input models with similar size, while the second one illustrates that the speed-up grows with increasing model size.

8.1.1. Experiment I: speed-up curve

For this experiment, we have used a set of 5 automatically generated Java programs with random structure but similar size and complexity. The source Java files range from 1 442 to 1 533 lines of code and the execution time of their sequential transformation ranges from 620s to 778s. The experiments were run on a set of identical *Elastic MapReduce* clusters provided by *Amazon Web Services*. All the clusters were composed of 10 EC2 instances of type *m1.large* (i.e. 2 vCPU, 7.5GB of RAM memory and 2 magnetic Hard Drives). Each execution of the transformation was launched in one of those clusters with a fixed number of nodes – from 1 to 8 – depending on the experiment. Each experiment has been executed 10 times for each model and number of nodes. In total 400 experiments have been executed summing up a total of 280 hours of computation (1 120 normalized instance hours[62]). For each execution, we calculate the distribution speed-up w.r.t. the same transformation on standard ATL running in a single node of the cluster.

Fig. 8 summarizes the speed-up results. The approach shows good performance for this transformation with an average speed-up between 2.5 and 3 on 8 nodes. More importantly, as it can be seen on the right-hand side, we see a quasi-

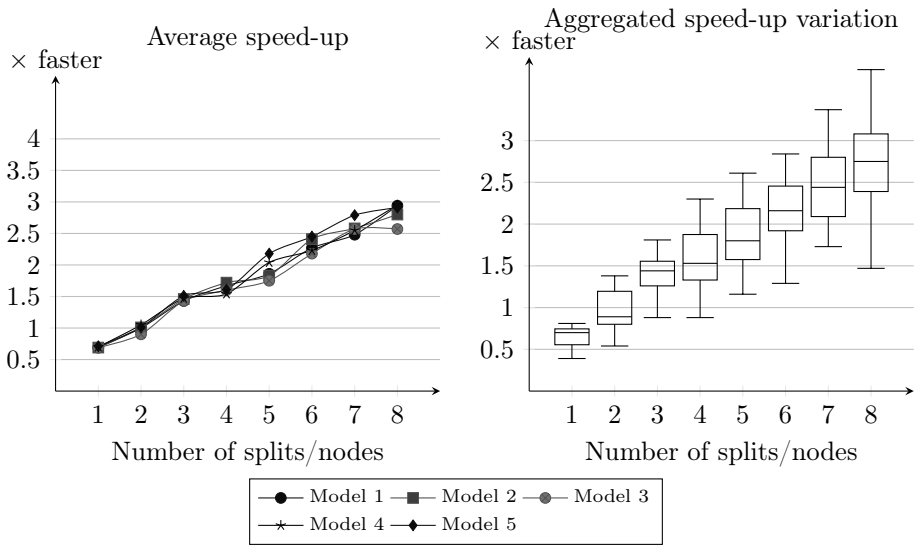


Figure 8: Speed-up obtained in experiment I

Table 6: Execution times and speed-up (between parentheses) per model

#	Size	Elts	Std. VM	Distributed VM using x nodes (time and speed-up)							
				1	2	3	4	5	6	7	8
1	∼4MB	20 706	244s	319s (×0.8)	165s (×1.5)	128s (×1.9)	107s (×2.3)	94s (×2.6)	84s (×2.9)	79s (×3.1)	75s (×3.3)
2	∼8MB	41 406	1 005s	1 219s (×0.8)	596s (×1.7)	465s (×2.2)	350s (×2.9)	302s (×3.3)	259s (×3.9)	229s (×4.4)	199s (×5.1)
3	∼16MB	82 806	4 241s	4 864s (×0.9)	2 318s (×1.8)	1 701s (×2.5)	1 332s (×3.2)	1 149s (×3.7)	945s (×4.5)	862s (×4.9)	717s (×5.9)
4	∼32MB	161 006	14 705s	17 998s (×0.8)	8 712s (×1.7)	6 389s (×2.3)	5 016s (×2.9)	4 048s (×3.6)	3 564s (×4.1)	3 050s (×4.8)	2 642s (×5.6)

linear speedup, with a very similar curve for all models under transformation. We naturally expect the speed-up curve to become sub-linear for larger cluster sizes and very unbalanced models. The variance among the 400 executions is limited as shown by the box-plots in the lower side.

8.1.2. Experiment II: size/speed-up correlation

To investigate the correlation between model size and speed-up we execute the transformation over 4 artificially generated Java programs with identical structure but different size (from 13 500 to 105 000 lines of code). Specifically, these Java programs are built by replicating the same imperative code pattern and they produce a balanced execution of the model transformation in the nodes of the cluster. This way, we abstract from possible load unbalance that would hamper the correlation assessment.

This time the experiments have been executed in a virtual cluster composed of 12 instances (8 slaves, and 4 additional instances for orchestrating Hadoop and HDFS services) built on top of OpenVZ containers running Hadoop 2.5.1. The hardware hosting the virtual cluster is a Dell PowerEdge R710 server, with two Intel® Xeon® X5570 processors at 2.93GHz (allowing up to 16 execution threads), 72 GB of RAM memory (1 066MHz), and two hard disks (at 15K rpm) configured in a hardware-controlled RAID 1.

As shown in Fig. 9 and Table 6, the curves produced by Experiment II are consistent with the results obtained from Experiment I, despite the different model sizes and cluster architectures. Moreover, as expected, larger models produce higher speed-ups: for longer transformations, the parallelization benefits of longer map tasks overtake the overhead of the MapReduce framework.

8.2. ATL-MR on NeoEMF/Column

In this experiment, we use NeoEMF/Column as a persistence backend, and evaluate its impact on the execution performance of ATL-MR. W.r.t the previous experimentation, we limit the amount of memory assigned to each map and reduce task to 2GB. This way, models do not fully fit in memory and the system is forced to rely on the persistence backend for model access during the transformation. In order to isolate the effect of the persistence backend from the

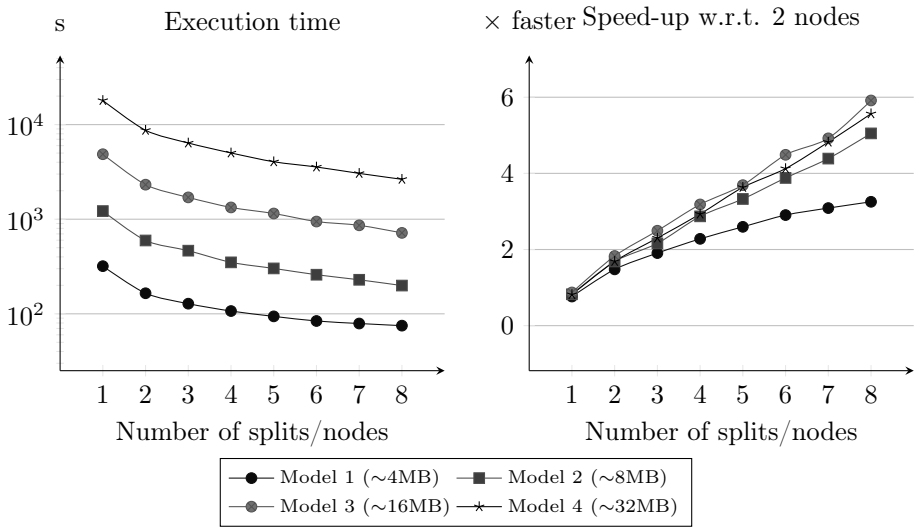


Figure 9: Execution times and speed-up on Experiment II

computational cost of the transformation logic, for this experiment, we use a simpler transformation, with linear-time complexity, i.e. *Class2Relational*. The experimentation shows that ATL-MR has also good performances for IO-bound transformations, that are not computationally expensive. The experiments have been executed in a virtual cluster of Docker containers running in a QEMU/KVM virtual machine executing Debian 9.3, with 32GB of RAM, 6 virtual CPUs, and paravirtualized hardware. Each Docker container runs Hadoop 2.7.3 and HBase 1.2.5. The virtual machine is hosted in a Fujitsu Primergy RX200 S8 server, equipped with two quad-core Intel® Xeon® Intel(R) Xeon(R) CPU E5-2609 v2 at 2.50GHz (thus allowing up to 8 execution threads), 48 GB of DDR3 RAM memory (1 333 MHz), and two hard disks (at 7 200 rpm) configured in a software-controlled RAID 1. The docker cluster replicating the environment setup is available online.

We use as input randomly generated models with diverse sizes. We make our random generator publicly available[10]. Its configuration, among other parameters, takes as input the model size and the density of references (i.e., the average number of references to be generated per property). The configuration also specifies an allowed deviation w.r.t. these parameters. In our experiment, we use an average density of 8 references per property and we allow a deviation of 10%. The generation of random models is seeded in order to make the experiments reproducible.

For each model size (10k, 20k, 30k, 40k, 50k, 100k elements), we generate three random models. We launch the transformation of each generated model on

[10]https://github.com/atlanmod/neoEMF-Instantiator

Table 7: Execution times and speed-up (between parentheses) per model

Elts	2	3	4	5	6	7	8
10 000	124s	90s (×1.38)	73s (×1.70)	71s (×1.75)	72s (×1.72)	77s (×1.61)	79s (×1.60)
20 000	228s	141s (×1.62)	120s (×1.90)	96s (×2.38)	92s (×2.48)	93s (×2.45)	93s (×2.45)
30 000	411s	223s (×1.84)	164s (×2.50)	144s (×2.85)	134s (×3.07)	113s (×3.64)	119s (×3.45)
40 000	751s	369s (×2.04)	262s (×2.87)	180s (×4.17)	161s (×4.66)	154s (×4.88)	140s (×5.36)
50 000	807s	396s (×2.04)	262s (×3.08)	188s (×4.29)	175s (×4.61)	155s (×5.21)	149s (×5.42)
100 000	–	–	859s	611s	485s	387s	380s

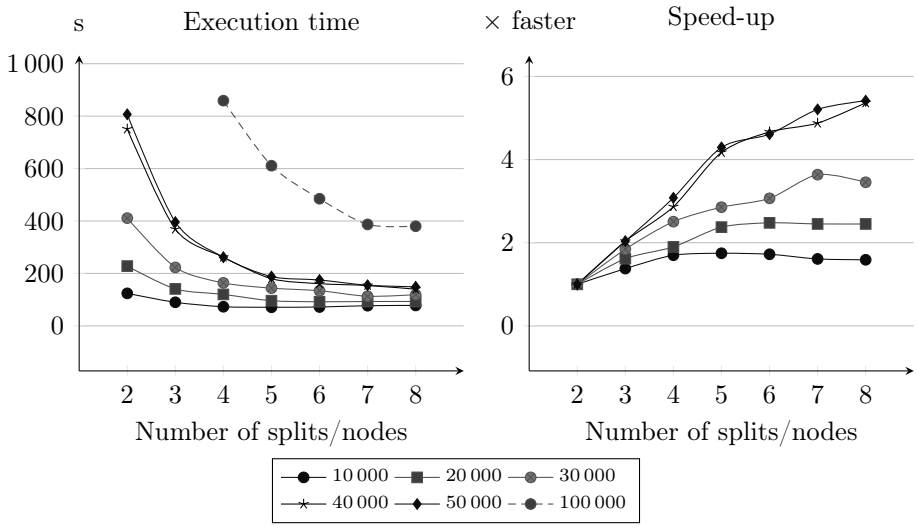


Figure 10: Execution times and speed-up of ATL-MR

subsets of the cluster, ranging from 2 to 8 nodes, for a total of 126 executions. Here we use a random data distribution strategy, while in [56] we report on an advanced greedy strategy.

Fig. 10 summarizes the execution time and speed-up results (exact values are listed in Table 7). The left-hand side plots the average execution time. The right-hand side shows the speed-up curves, with an average speed-up between 1.6 and 5.4. The transformation of the largest model (100 000 elements) requires at least 4 nodes (thus we do not calculate speed-up for this case).

Differently from the XMI case, for smaller models (up to 30 000 elements), the speed-up curve has a logarithmic shape, while it becomes quasi-linear for larger models. Besides the general scalability of the HBase back-end, two factors are impacting this result. First, the computational complexity of the Class2Relational transformation is lower than the ControlFlow2DataFlow transformation. Second,

for smaller models, the ratio of the Hadoop environment setup time to the overall execution time is very important (15% to 20%).

8.3. Discussion

In our experimentations, we evaluated the performance of ATL-MR on top of XMI and NeoEMF/Column. We have shown that ATL-MR scales for two use cases that are representative of two application classes. 1) Controlflow2Dataflow represents model transformations that are CPU-bound (i.e. perform complex computations), despite requiring extensive reads to the source model. In this case, the gain in parallelizing the computation outweighs the cost of accessing big portions of the source model in several nodes. 2) Class2Relational represents model transformation that despite being IO-bound (i.e. they require very simple computation), perform localized access to small parts of the source model for each rule application. In this case, source-model access can be efficiently parallelized on HBase.

With the XMI backend, each map node needs to load the whole model in memory, hampering the transformation of models that do not fit in memory. Moreover, since concurrent writes are not allowed on files, it is not possible to parallelize the reduce phase using ATL-MR on top of XMI. Additionally, for short transformation times (less than 3 minutes in our experimentation), the gain obtained by parallelization cannot compensate the overhead of the MapReduce framework. Unfortunately, it is difficult to quantify the MapReduce framework's overhead before execution, as it depends on the execution environment and the models' size. The bigger the model, the more time it takes to split it and shuffle it. We can conclude that ATL-MR on XMI is suitable for running complex transformations on models that fit in memory, for example, transformations of models requiring extensive data analysis.

In order to solve the issues above, we proposed NeoEMF/Column. Thanks to the lazy-loading mechanism, we were able to load in each node only the model elements that are necessary to compute the transformation of the assigned split. Besides, NeoEMF/Column allows concurrent read/write, enabling the parallelization of the reduce phase. According to existing benchmarks for top NoSQL databases [63], HBase is well-suited for read-mostly workloads but not insert-mostly workloads. As a result, when dealing with transformations requiring extensive writes on target models, some performance drops can be noticed. Finally, it is worth to mention that the same models require more space when stored in NeoEMF/Column, compared to XMI. This is because of HBase internals. To deliver high-availability and fault-tolerance, HBase relies on table replication, table snapshots, and rows versions. These features impact heavily the size of the tables on disk. One way to reduce the size is by using data compression techniques, but this impacts drastically the read and write latency. We can conclude that ATL-MR on NeoEMF/Column is suitable for running MTs on large models with balanced read/write workloads. Most of the model transformations that are present in the ATL-Zoo [64] can be considered as read-write workload balanced.

Table 8: Yarn Options

Option	Description
yarn.nodemanager.resource.memory-mb	Amount of physical memory for all containers
yarn.nodemanager.container-monitor.interval-ms	How often to monitor containers
yarn.nodemanager.resource.cpu-vcores	Number of CPU cores that can be allocated for containers
yarn.nodemanager.disk-health-checker.max-disk-utilization-per-disk-percentage	The maximum percentage of disk before failure

Choosing the right number of splits has a significant impact on the global performance. Having many splits means that the time that is taken to process each split will be small compared to the time to process the whole input. On the other hand, if splits are too small, then the overhead of managing the splits and creating map tasks for each one of them may dominate the total job execution time. In our case, we observed better results where the number of splits matches the number of available workers. In other words, while configuring Distributed ATL, the number of lines per split should be set to $\frac{model\ size}{available\ nodes}$.

Furthermore, one should be careful with Hadoop (Yarn) configuration options. In the case of memory or time-consuming model transformations, it is required to set up correctly these options in order to avoid the system's failure. Table 8 depicts some of these options. Note that node manager resources are shared among all the containers running under it. The number of containers is configurable as well (see Table 8).

Even though ATL supports both out-place and in-place model transformations –through refinement mode–, in our prototype, only out-place model transformations are supported. Moreover, we do not handle ATL imperative code blocks. We argue that, although imperative blocks simplify the expression of complex transformations, most of the transformation logic can be specified using declarative-only ATL. As for lazy rules and unique lazy rules they are currently not parallelized, but handled in the same way as in EMFTVM, i.e., they are executed in a final step. Finally, ATL does not support recursive rule applications (i.e., further transformation of target elements), neither does ATL-MR.

9. Conclusion and Future Work

In this paper, we exploited the recent emergence of systems and programming models for distributed and parallel processing to leverage the distributed transformation and persistence of VLMs. In particular, we relied on a well-known distributed programming model, MapReduce, in order to enable the distributed execution of model transformation in a reliable and fault-tolerant manner. We also adopted NoSQL databases as a new solution for persisting VLMs. We showed

that relational model transformation with languages like ATL is a problem that fits in the MapReduce execution model. As a proof of concept, we introduced a semantics for ATL distributed execution on MapReduce. We experimentally showed the good scalability of our solution. Thanks to our publicly available execution engine, users may exploit the availability of MapReduce clusters on the Cloud to run model transformations in a scalable and fault-tolerant way.

Moreover, we exposed some limitations in standard persistence backends in EMF and proposed a solution for transparent and decentralized model persistence and manipulation in EMF on top of HBase. We also defined the minimal set of ACID properties that model access has to guarantee for a consistent and correct model transformation. We achieve this by cross-analyzing the execution semantics of ATL, especially atomic model manipulation operations (as coming from the MOF specification), against the MOF Reflection API. We intend to improve the efficiency of our distributed transformation engine by exploring the following lines:

- Reducing the number of ⟨key,value⟩ pairs transmitted between the *Local Match-Apply* and the *Global Resolve* phases can improve the time to perform the *shuffling* phase. In future work, we want to cut down the number of transmitted elements based on the static analysis of the bindings of the transformation rules as well as inter-rules dependency. An example of tracing information that can be omitted, are the ones involving model elements that will not be resolved by any other target element.

- In the current version of ATL-MR, the fault-tolerance relies completely on failure-management as provided by Yarn. Since in ATL-MR, output models are directly written to the persistence store, we can extend ATL-MR with a centralized tracking system that helps the transformation engine recovering from the previous state and carry on with the transformation.

- Some relational MT languages like ETL and QVT/Relations share most of their properties with ATL. For instance, ETL and QVT also rely on tracing information for resolving target elements. The ETL also runs in two steps as in ATL. In future work, we plan to investigate the generalization of our approach to these languages in detail. Likewise, we would like to adapt our approach to other, and more recent, distributed programming frameworks, in order to investigate which framework is most suitable to what MT scenario.

- OCL is a central component in model transformation languages. It is used to specify guards, query models, target sets, etc.. Collection operations are the most computationally expensive operations in OCL. One way to break down their complexity is by providing parallelism at collection operations level. We believe that providing a distributed OCL engine can alleviate the development of not only distributed MT but also other model-based solutions, such as distributed query engines. We also believe that it would answer some open challenges in the MDE community such

as streaming model transformations, pipelining and scheduling distributed model transformations.

Acknowledgments

This work is partially supported by the MONDO project, EU Seventh Framework programme No. ICT-611125; and the MegaM@Rt2 project, which has received funding from the Electronic Component Systems for European Leadership Joint Undertaking under grant agreement No. 737494. This Joint Undertaking receives support from the European Union's Horizon 2020 research and innovation programme and Sweden, France, Spain, Italy, Finland, and Czech Republic.

References

[1] R. Volk, J. Stengel, F. Schultmann, Building Information Modeling (BIM) for Existing Buildings: Literature Review and Future Needs, Automation in Construction 38 (0) (2014) 109–127.

[2] H. Bruneliére, J. Cabot, G. Dupé, F. Madiot, MoDisco: A Model Driven Reverse Engineering Framework, Information and Software Technology 56 (8) (2014) 1012–1032.

[3] B. Dominic, Towards Model-Driven Engineering for Big Data Analytics – An Exploratory Analysis of Domain-Specific Languages for Machine Learning, in: Proceedings of The 47th Hawaii International Conference on System Sciences (HICSS), 2014, pp. 758–767.

[4] P. Baker, S. Loh, F. Weil, Model-driven engineering in a large industrial context: Motorola case study, in: Model Driven Engineering Languages and Systems, Springer, 2005, pp. 476–491.

[5] D. S. Kolovos, R. F. Paige, F. A. Polack, The grand challenge of scalability for model driven engineering, in: Models in Software Engineering, Springer, 2009, pp. 48–53.

[6] J. Hutchinson, J. Whittle, M. Rouncefield, S. Kristoffersen, Empirical assessment of mde in industry, in: Proceedings of the 33rd International Conference on Software Engineering, ACM, 2011, pp. 471–480.

[7] A. Forward, T. C. Lethbridge, Problems and Opportunities for Model-centric Versus Code-centric Software Development: A Survey of Software Professionals, in: Proceedings of the 2008 International Workshop on Models in Software Engineering, MiSE '08, ACM, New York, NY, USA, 2008, pp. 27–32.

[8] M. Laakso, A. Kiviniemi, The IFC standard: A Review of History, Development, and Standardization, Information Technology, ITcon 17 (9) (2012) 134–161.

[9] D. Durisic, M. Staron, M. Tichy, J. Hansson, Evolution of long-term industrial meta-models – an automotive case study of autosar, in: Software Engineering and Advanced Applications (SEAA), 2014 40th EUROMICRO Conference on, 2014, pp. 141–148.

[10] D. Durisic, M. Staron, M. Tichy, J. Hansson, Quantifying Long-Term Evolution of Industrial Meta-Models-A Case Study, in: Software Measurement and the International Conference on Software Process and Product Measurement (IWSM-MENSURA), 2014 Joint Conference of the International Workshop on, IEEE, 2014, pp. 104–113.

[11] J. Dean, S. Ghemawat, Mapreduce: Simplified data processing on large clusters, Communications of the ACM 51 (1) (2008) 107–113. doi:10.1145/1327452.1327492.

[12] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, G. Czajkowski, Pregel: A System for Large-scale Graph Processing, in: Proceeding of the 2010 ACM SIGMOD International Conference on Management of Data, SIGMOD '10, ACM, Indianapolis, Indiana, USA, 2010, pp. 135–146.

[13] C. Clasen, M. Didonet Del Fabro, M. Tisi, Transforming Very Large Models in the Cloud: a Research Roadmap, in: First International Workshop on Model-Driven Engineering on and for the Cloud, Springer, Copenhagen, Denmark, 2012.

[14] F. Jouault, F. Allilaire, J. Bézivin, I. Kurtev, ATL: A Model Transformation Tool, Science of Computer Programming 72 (1-2) (2008) 31–39, special Issue on 2nd issue of experimental software and toolkits (EST).

[15] A. Benelallam, A. Gómez, M. Tisi, J. Cabot, Distributed Model-to-model Transformation with ATL on MapReduce, in: Proceedings of the 2015 ACM SIGPLAN International Conference on Software Language Engineering, SLE 2015, ACM, New York, NY, USA, 2015, pp. 37–48.

[16] A. Gómez, A. Benelallam, M. Tisi, Decentralized Model Persistence for Distributed Computing, in: Proceedings of 3rd BigMDE Workshop, Vol. 1406, CEUR Workshop Proceedings, 2015.

[17] H. Ehrig, K. Ehrig, U. Prange, G. Taentzer, Fundamentals of Algebraic Graph Transformation, Springer-Verlag, 2006.

[18] G. Mezei, T. Levendovszky, T. Meszaros, I. Madari, Towards Truly Parallel Model Transformations: A Distributed Pattern Matching Approach, in: IEEE EUROCON 2009, IEEE, 2009, pp. 403–410.

[19] B. Izsó, G. Szárnyas, I. Ráth, D. Varró, IncQuery-D Incremental Graph Search in the Cloud, in: Proceedings of the Workshop on Scalability in MDE, BigMDE '13, ACM, New York, NY, USA, 2013, pp. 4:1–4:4.

[20] C. Krause, M. Tichy, H. Giese, Implementing graph transformations in the bulk synchronous parallel model, in: International Conference on Fundamental Approaches to Software Engineering, Springer, 2014, pp. 325–339.

[21] L.-D. Tung, Z. Hu, Towards systematic parallelization of graph transformations over pregel, International Journal of Parallel Programming (2015) 1–20.

[22] L. Burgueño, J. Troya, M. Wimmer, A. Vallecillo, On the Concurrent Execution of Model Transformations with Linda, in: Proceeding of the First Workshop on Scalability in MDE, BigMDE '13, ACM, New York, NY, USA, 2013, pp. 3:1–3:10.

[23] L. Burgueño, E. Syriani, M. Wimmer, J. Gray, A. Moreno Vallecillo, LinTraP: Primitive Operators for the Execution of Model Transformations with LinTra, in: Proceedings of 2nd BigMDE Workshop, Vol. 1206, CEUR Workshop Proceedings, 2014.

[24] M. Tisi, S. Martinez, H. Choura, Parallel execution of atl transformation rules, in: Model-Driven Engineering Languages and Systems, Springer, 2013, pp. 656–672.

[25] G. Imre, G. Mezei, Parallel Graph Transformations on Multicore Systems, in: Multicore Software Engineering, Performance, and Tools, Vol. 7303 of LNCS, Springer, 2012, pp. 86–89.

[26] G. Bergmann, I. Ráth, D. Varró, Parallelization of graph transformation based on incremental pattern matching, Electronic Communications of the EASST Vol. 18.
URL http://journal.ub.tu-berlin.de/index.php/eceasst/article/view/265

[27] G. Bergmann, A. Horváth, I. Ráth, Incremental evaluation of model queries over EMF models, International Conference on Model Driven Engineering Languages and Systems.
URL http://link.springer.com/chapter/10.1007/978-3-642-16145-2_6

[28] Z. Ujhelyi, G. Bergmann, Á. Hegedüs, Á. Horváth, B. Izsó, I. Ráth, Z. Szatmári, D. Varró, Emf-incquery: An integrated development environment for live model queries, Sci. Comput. Program. 98 (2015) 80–99. doi:10.1016/j.scico.2014.01.004.
URL https://doi.org/10.1016/j.scico.2014.01.004

[29] G. Bergmann, D. Horváth, Á. Horváth, Applying incremental graph transformation to existing models in relational databases, in: Graph Transformations, Springer, 2012, pp. 371–385.

[30] F. Jouault, M. Tisi, Towards incremental execution of atl transformations, in: Theory and Practice of Model Transformations, Springer, 2010, pp. 123–137.

[31] H. Giese, R. Wagner, From model transformation to incremental bidirectional model synchronization, Software & Systems Modeling 8 (1) (2009) 21–43.

[32] G. Bergmann, I. Dávid, Á. Hegedüs, Á. Horváth, I. Ráth, Z. Ujhelyi, D. Varró, Viatra 3: A reactive model transformation platform, in: International Conference on Theory and Practice of Model Transformations, Springer, 2015, pp. 101–110.

[33] I. Dávid, I. Ráth, D. Varró, Streaming model transformations by complex event processing, in: International Conference on Model Driven Engineering Languages and Systems, Springer, 2014, pp. 68–83.

[34] S. Martínez, M. Tisi, R. Douence, Reactive model transformation with atl, Science of Computer Programming 136 (2017) 1–16.

[35] R. Chaiken, B. Jenkins, P.-Å. Larson, B. Ramsey, D. Shakib, S. Weaver, J. Zhou, SCOPE: Easy and Efficient Parallel Processing of Massive Data Sets, Proceedings of the VLDB Endowment 1 (2) (2008) 1265–1276.

[36] C. Olston, B. Reed, U. Srivastava, R. Kumar, A. Tomkins, Pig latin: a not-so-foreign language for data processing, in: Proceedings of the 2008 ACM SIGMOD international conference on Management of data, ACM, 2008, pp. 1099–1110.

[37] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, R. Murthy, Hive: a warehousing solution over a map-reduce framework, Proceedings of the VLDB Endowment 2 (2) (2009) 1626–1629.

[38] M. Isard, M. Budiu, Y. Yu, A. Birrell, D. Fetterly, Dryad: distributed data-parallel programs from sequential building blocks, in: ACM SIGOPS Operating Systems Review, Vol. 41, ACM, 2007, pp. 59–72.

[39] N. Amálio, J. de Lara, E. Guerra, Fragmenta: A theory of fragmentation for MDE, in: The ACM/IEEE 18th International Conference on Model Driven Engineering Languages and Systems (MODELS), 2015, 2015, pp. 106–115.

[40] Markus Scheidgen, EMF fragments (2014).
URL `https://github.com/markus1978/emf-fragments/wiki`

[41] A. Garmendia, E. Guerra, D. S. Kolovos, J. de Lara, EMF Splitter: A Structured Approach to EMF Modularity, XM@ MoDELS, 3rd Extreme Modeling Workshop 1239 (2014) 22–31.

[42] CDO Model Repository (2014).
URL `http://www.eclipse.org/cdo/`

[43] M. Scheidgen, A. Zubow, J. Fischer, T. H. Kolbe, Automated and Transparent Model Fragmentation for Persisting Large Models, in: 15th International Conference on Model Driven Engineering Languages and Systems, Springer-Verlag, 2012, pp. 102–118.

[44] J. E. Pagán, J. S. Cuadrado, J. G. Molina, Morsa: A Scalable Approach for Persisting and Accessing Large Models, in: 14th International Conference on Model Driven Engineering Languages and Systems, Springer-Verlag, 2011, pp. 77–92.

[45] J. E. Pagán, J. G. Molina, Querying large models efficiently, Information and Software Technology 56 (6) (2014) 586 – 622.

[46] K. Barmpis, D. S. Kolovos, Comparative analysis of data persistence technologies for large-scale models, in: Proceedings of the 2012 Extreme Modeling Workshop, XM '12, ACM, New York, NY, USA, 2012, pp. 33–38.

[47] MongoDB Inc., MongoDB (2016).
URL `https://www.mongodb.com/`

[48] Bryan Hunt, MongoEMF (2014).
URL `https://github.com/BryanHunt/mongo-emf/`

[49] T. Horn, The TTC 2013 Flowgraphs Case, arXiv preprint arXiv:1312.0341.

[50] Object Management Group, Object Constraint Language, OCL, URL: `http://www.omg.org/spec/OCL/` (May, 2016).

[51] K. Lano, S. Kolahdouz-Rahimi, Model-transformation design patterns, IEEE Transactions on Software Engineering 40 (12) (2014) 1224–1259. `doi:10.1109/TSE.2014.2354344`.

[52] Apache Software Foundation, Apache Hadoop, URL: `http://hadoop.apache.org/` (May, 2016).

[53] Apache Software Foundation, Apache Hadoop Distributed File System (HDFS), URL: `https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html` (May, 2016).

[54] C. Gomes, B. Barroca, V. Amaral, Classification of Model Transformation Tools: Pattern Matching Techniques, in: Proceedings of 17th International Conference Model-Driven Engineering Languages and Systems, Springer International Publishing, 2014, pp. 619–635.

[55] M. R. Fellows, J. Guo, C. Komusiewicz, R. Niedermeier, J. Uhlmann, Graph-based Data Clustering with Overlaps, Discrete Optimization 8 (1) (2011) 2–17.

[56] A. Benelallam, M. Tisi, J. Sánchez Cuadrado, J. de Lara, J. Cabot, Efficient Model Partitioning for Distributed Model Transformations, in: Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering, SLE 2016, ACM, New York, NY, USA, 2016.

[57] M. Kuramochi, G. Karypis, Frequent subgraph discovery (2001). doi:10.1109/ICDM.2001.989534.

[58] E. Brewer, CAP twelve years later: How the "rules" have changed, Computer 45 (2) (2012) 23–29.

[59] Object Management Group, XML Metadata Interchange, URL: http://www.omg.org/spec/XMI/ (May, 2016).

[60] G. Daniel, G. Sunyé, A. Benelallam, M. Tisi, Y. Vernageau, A. Gómez, J. Cabot, NeoEMF: A multi-database model persistence framework for very large models, Sci. Comput. Program. 149 (2017) 9–14.

[61] J. Lin, M. Schatz, Design patterns for efficient graph algorithms in mapreduce, in: Proceedings of the Eighth Workshop on Mining and Learning with Graphs, MLG '10, ACM, New York, NY, USA, 2010, pp. 78–85.

[62] Amazon Web Services, Inc., Amazon EMR FAQs, URL: http://aws.amazon.com/elasticmapreduce/faqs (May, 2016).

[63] Onepoint, https://www.datastax.com/wp-content/themes/datastax-2014-08/files/NoSQL_Benchmarks_EndPoint.pdf (2015).

[64] The atl transformation zoo, http://www.eclipse.org/atl/atlTransformations/ (2014).